Verified Post-Quantum Cryptography

Verifying The McEliece Cryptosystem With Refinement Types

Tom Arnold
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
tca4384@rit.edu

Abstract

New cryptosystems are being developed and standardized to resist attacks from quantum computers; one such cryptosystem is the McEliece cryptosystem which owes its security to coding theory and binary Goppa codes. Implementation of the McEliece cryptosystem involves linear algebra, coding theory, and arithmetic with polynomials of binary Galois field elements.

Cryptosystems are one class of software where the absence of defects is highly desirable; this is where formal verification is useful. Formal verification can help prove mathematically the correct implementation of these systems.

We present a study of formal verification applied to the domain of post-quantum cryptography by implementing the McEliece cryptosystem in LiquidHaskell and verifying it using refinement types. This includes all of the supporting mathematics required for the cryptosystem including linear algebra, coding theory, polynomial arithmetic, and binary Galois field arithmetic. As part of this study we show that refinement types are a useful tool for reducing the chance of security breaking bugs during implementation of a cryptosystem.

I. Introduction

Widely used public key cryptosystems are vulnerable to attacks from quantum computers. These cryptosystems are based on one-way functions for which there is no efficient solution to calculate the inverse of the function; the security of the system is built on this fact. Quantum computing has introduced new algorithms which can efficiently compute solutions to some of these problems. As a result, cryptographers are working to implement new cryptosystems based on problems that are not efficiently solvable by quantum or digital computers [1].

The McEliece cryptosystem is one such system that dates back to the 70s when it was proposed by Robert McEliece. The system is based on coding theory and to break it an attacker would have to solve the general decoding problem which is NP-complete [5]. A modern variant of McEliece called Classic McEliece is one of the finalists in the NIST Post-Quantum standardization effort [2].

Formal verification can be used to prove the absence of bugs in a way that automated or manual testing cannot. Typically such verification is only done for software which must be held to a high standard. Cryptography is one such field where even simple errors can have severe real-world consequences [3].

In the following paper we show how a post-quantum cryptosystem (McEliece) can be implemented and formally verified using refinement types with LiquidHaskell [4]. We also examine the effort involved in performing such verification so as to better understand the costs and benefits involved.

II. Post-Quantum Cryptography

Post-quantum cryptography is a new field of cryptography focused on the creation of cryptosystems which are secure despite the existence of quantum computers. Quantum computers are able to natively execute quantum algorithms which can solve some problems much more efficiently than a classical computer could. This is a problem for cryptography because public-key cryptography is built on one-way functions, i.e. functions that are easy to compute in one direction but difficult to compute the reverse of, and for some problems there exist quantum algorithms that break this assumption. The obvious example of this concern is Shor's algorithm and the integer factoring problem which is used by the famous RSA public-key cryptosystem.

Although large-scale quantum computers do not yet exist, it seems likely that they will in the near future and replacing our existing pre-quantum cryptosystems will take years. Additionally anything encrypted today with pre-quantum cryptography could be broken once someone is able to develop a large-scale quantum computer. As a result of these concerns, NIST is currently running a post-quantum cryptography competition with the goal of standardizing several post-quantum cryptosystems so that government and industry can begin requiring and using them.

One of the cryptosystems being standardized by NIST is Classic McEliece, a key-encapsulation mechanism (KEM) built on the Niederreiter public-key cryptosystem which is a variant of the original McEliece public-key cryptosystem proposed by Robert McEliece in the 70s.

While the McEliece and Niederreiter schemes are quite old, they are based on a problem (i.e. general decoding) which does not have a quantum algorithm and is NP-complete, therefore they are post-quantum safe. These systems have not gotten much attention from cryptographers until recently

due to their large key sizes. Both systems use large matrices of bits for their private and public keys, and while they work with any linear code so far binary Goppa codes are the only one that has proven to be secure. The primary difference between the two systems is that Niederreiter uses a parity check matrix instead of a generator matrix as its public key and as a result the ciphertext is a syndrome vector instead of a garbled codeword. An advantage of the Niederreiter system is that the parity matrix can be represented in standard form $H = (I|M)$ which allows the key size to be halved. The Niederreiter system can also be used for digital signatures unlike the original McEliece system.

Because of their similarity with each other and given that a modern post-quantum cryptosystem (Classic McEliece) is based on one of them, the McEliece and Niederreiter schemes are an accessible starting point for formal verification; they are relatively simple but also relevant.

III. Verification Techniques

Formal verification can be used to prove the absence of certain types of bugs in a program. This technique typically uses some kind of theorem prover to mathematically prove certain properties about a program. One way to apply this technique is to integrate it into the type system of a programming language in the form of refinement types.

Refinement types are a way to refine a type to consist of more specific values. For example, suppose we have a function f which accepts an integer parameter x and performs a runtime assertion to check that the parameter is even. Instead of doing this check at runtime we could refine the type of x from integer to even integer, i.e. integers divisible by two or $x : \{Int|x \bmod 2 = 0\}$ which reads as "x is an integer such that x is divisible by 2". More complicated refinements are possible, including refining the type of a parameter relative to another parameter, or refining the return type of a function based on its inputs.

Haskell is a functional programming language with an advanced type system. Haskell is typically used as a research language for programming language theory. Code written in Haskell is strongly-typed and the language provides many features to make it convenient to work with this system; for example Haskell allows developers to implement custom number types by defining function to convert to/from integers and to perform arithmetic. A plugin to Haskell called Liquid Haskell provides refinement types in the form of comment annotations that the plugin parses during compilation. After parsing the code, Liquid Haskell generates constraints and invokes the Z3 theorem prover to determine if the constraints are satisfiable; if they are not then this usually indicates a flaw in the program logic and an error message is returned to the developer. In the next sections we will discuss in detail what refinement types can be used for.

A. Static Types

B. Refinement Types

C. Size-Aware API

D. Termination Checking

IV. Implementation

Having discussed refinement types, we will now dive into implementing and verifying the McEliece and Niederreiter post-quantum cryptosystems in Haskell. The full source-code is available at https://bitbucket.org/Tom9729/csci788/src/master/ and can be compiled using the Haskell Tool Stack https://www.haskellstack.org/.

A. Support Code

Many libraries exist to do matrix and Galois field math in Haskell, but none of them are verified with refinement types. While Liquid Haskell provides a mechanism to introduce refinements to an external library, for this project we chose to simply re-implement the required math for the McEliece and Niederreiter schemes.

1) Matrices:

2) Binary Matrix Inversion:

3) Matrix Right-Inverse:

4) Polynomials:

5) Galois Fields:

6) Extended Euclidean Algorithm (EEA): The Extended Euclid Algorithm (also known as EEA) computes the greatest common divisor d of two numbers and returns it as well as the coefficients xy of Bezout's identity such that $d = a \times x + b \times y$. This is typically used in cryptography to find the multiplicative inverse of a Galois field element.

Implementation of the EEA in Haskell is a straightforward conversion from textbook pseudocode iteration to recursion. Of note is that because of Haskell's type system we are able to implement the EEA once for numbers, polynomials, and field elements! Here refinement types provide us with two additional checks beyond what the Haskell type system provides.

First we refine the second parameter to be non-zero because we perform a division by it and that would result in a division-by-zero runtime error. Second we refine the inner recursive loop go to have a termination condition. As long as $b0 > r$ the remainder of $b0/r$ is decreasing, and since $r = 0$ is our base case we have shown inductively that the function will always terminate.

```
{-@ extEuclidAlg :: (Integral a)
                 => a:a
                 -> b:{a | b /= 0}
                 -> (a, a, a) @-}
extEuclidAlg a b =
  let (q, r) = a `quotRem` b
  in
    go 0 1 1 0 a b q r
  where
    {-@ go :: (Integral a)
           => t0:a -> t:a -> s0:a -> s:a -> a0:a -> b0:a ↩
             -> q:a
           -> r:{a | b0 > r}
           -> (a,a,a) / [r] @-}
```

```
go t0 t s0 s a0 b0 q r =
  if r == 0 then (b0, s, t)
  else
    let t' = t0 - q * t
        s' = s0 - q * s
        (q',r') = b0 `quotRem` r
    in
      go t t' s s' b0 r q' r'
```

7) Modified EEA:

B. McEliece Cryptosystem

1) Key Generation:

2) Encryption:

3) Decryption:

C. Niederreiter Cryptosystem

1) Key Generation:

2) Encryption:

3) Decryption:

V. Results

Implemented McEliece and Niederreiter cryptosystems in Haskell and verified with refinement types. Matrix and vector math including binary matrix inversion. Univariate polynomial math and binary Galois fields. Extended Euclid Algorithm. Patterson's decoding algorithm for Goppa codes. Toy examples of McEliece and Niederreiter encryption/decryption. All code verified with refinement types except for GF2 reduction and polynomial division which were not verified to terminate. Size-aware API for vectors and matrices provides bounds checking and prevents off-by-one errors. Recursive functions guaranteed to terminate. Avoid divide by zero by requiring non-zero parameters. Refinement types can eliminate certain classes of errors with only a modest increase of implementation complexity and are a useful tool for cryptography.

VI. Related Work

This project was inspired by MSR-INRIA's Project Everest which is implementing a formally verified TLS stack (https://project-everest.github.io/).

VII. Conclusion

Refinement types have been a useful tool for implementing McEliece variants during this project. In many cases the refinements were simply assertions that would normally be checked at runtime, lifted into compile-time checks. Adding refinements was generally a one-time development cost that paid for itself when editing the code later. Combined with a good test suite (this project used doctests) this strategy could be used to build high-assurance systems.

The main challenge of working with refinement types was proving properties about custom data types, e.g. proving that dividing by a non-zero polynomial was safe, or proving that adding binary field elements of the same degree would always produce an element of a lower degree. Refinements also slightly increased the compile time for the project, although upgrading to the latest Z3 theorem prover noticeably improved this.

References

[1] Risse, T. (2011). How SAGE Helps To Implement Goppa Codes and The McEliece Public Key Crypto System.

[2] Computer Security Division. Post-quantum cryptography: CSRC. Retrieved February 24, 2022, from https://csrc.nist.gov/Projects/post-quantum-cryptography

[3] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17). Association for Computing Machinery, New York, NY, USA, 1789–1806. DOI:https://doi.org/10.1145/3133956.3134043

[4] Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-jones, S. (2014). Refinement types for Haskell. ICFP 2014.

[5] McEliece, R.J. (1978). A public key cryptosystem based on algebraic coding theory.